```
# Prime Automata Lab — ULTIMATE EDITION with Graph, Topology & Fourier
# ---------------------------------------------------------------------

import math
import numpy as np
import numpy as np
from scipy import special
from typing import Any
import numba as nb
import matplotlib.pyplot as plt
from scipy import ndimage
from scipy.fft import fft2, fftshift
from scipy.ndimage import label, distance_transform_edt
from scipy.spatial import Delaunay
from sklearn.neighbors import NearestNeighbors
from collections import Counter
import networkx as nx

# Colab widgets setup
try:
    import ipywidgets as widgets
    from IPython.display import display, clear_output
    try:
        from google.colab import output as _colab_output
        _colab_output.enable_custom_widget_manager()
    except Exception:
        pass
except Exception as e:
    raise RuntimeError("ipywidgets is required. In Colab: !pip install ipywidgets; then rerun.") from
e



# ------------------------------
# Primes & helpers
# ------------------------------
def prime_sieve(limit: int) -> np.ndarray:
    if limit < 1:
        return np.array([False, False], dtype=bool)
    sieve = np.ones(limit + 1, dtype=bool)
    sieve[:2] = False
    for i in range(2, int(limit ** 0.5) + 1):
        if sieve[i]:
            sieve[i * i :: i] = False
    return sieve
```

```
def mask_from_values(vals, limit):
    m = np.zeros(limit + 1, dtype=bool)
    if len(vals):
        m[np.array(list(vals), dtype=int)] = True
    return m

def is_power_of_two(n: int) -> bool:
    return n > 0 and (n & (n - 1)) == 0

def sumsq_digits(n: int) -> int:
    s = 0
    while n:
        n, d = divmod(n, 10)
        s += d * d
    while n != 1 and n not in seen:
        if n in _happy_cache:
            return _happy_cache[n]
        seen.add(n)
        n = sumsq_digits(n)
    res = (n == 1)
    for v in seen:
        _happy_cache[v] = res
    return is_happy

def is_semiprime(n: int, primes_list) -> bool:
    cnt = 0
    temp = n
    for p in primes_list:
        if p * p > temp: break
        while temp % p == 0:
            cnt += 1
            temp //= p
            if cnt > 2: return False
    if temp > 1: cnt += 1
    return cnt == 2


# ----------------------------
# Prime-type masks
# ----------------------------
ALL_FILTERS = [
    "all","odd","twin","cousin","sexy",
    "safe","sophie_germain",
```

```
    "gaussian","eisenstein",
    "mersenne","fermat",
    "palindromic","emirp","circular",
    "left_truncatable","right_truncatable",
    "balanced","strong","weak",
    "happy","additive",
    "factorial_pm1","primorial_pm1",
    "proth","pierpont","wagstaff",
    "superprime","chen",
]

FERMAT_SET = {3, 5, 17, 257, 65537}

def get_filter_mask(name: str, prime_mask: np.ndarray):
    limit = len(prime_mask) - 1
    primes = np.flatnonzero(prime_mask)
    pset  = set(primes)

    if name == "all":
        return prime_mask.copy()
    if name == "odd":
        return mask_from_values([p for p in primes if p % 2 == 1], limit)
    if name == "twin":
        S = set()
        for p in primes:
            if p + 2 in pset: S.update((p, p + 2))
            if p - 2 in pset: S.update((p, p - 2))
        return mask_from_values(S, limit)
    if name == "cousin":
        S = set()
        for p in primes:
            if p + 4 in pset: S.update((p, p + 4))
            if p - 4 in pset: S.update((p, p - 4))
        return mask_from_values(S, limit)
    if name == "sexy":
        S = set()
        for p in primes:
            if p + 6 in pset: S.update((p, p + 6))
            if p - 6 in pset: S.update((p, p - 6))
        return mask_from_values(S, limit)
    if name == "safe":
        return mask_from_values([p for p in primes if ((p - 1) // 2) in pset], limit)
    if name == "sophie_germain":
        return mask_from_values([p for p in primes if 2 * p + 1 in pset], limit)
```

```
if name == "gaussian":
    return mask_from_values([2] + [p for p in primes if p % 4 == 3], limit)
if name == "eisenstein":
    return mask_from_values([p for p in primes if p % 3 == 2], limit)
if name == "mersenne":
    return mask_from_values([p for p in primes if is_power_of_two(p + 1)], limit)
if name == "fermat":
    return mask_from_values([p for p in primes if p in FERMAT_SET], limit)
if name == "palindromic":
    return mask_from_values([p for p in primes if str(p) == str(p)[::-1]], limit)
if name == "emirp":
    S = set()
    for p in primes:
        rp = int(str(p)[::-1])
        if rp != p and rp in pset:
            S.update((p, rp))
    return mask_from_values(S, limit)
if name == "circular":
    S = set()
    for p in primes:
        s = str(p)
        if len(s) > 1 and any(d in s for d in "024568"):
            continue
        ok = True
        for i in range(len(s)):
            r = int(s[i:] + s[:i])
            if r not in pset:
                ok = False
                break
        if ok:
            for i in range(len(s)):
                S.add(int(s[i:] + s[:i]))
    return mask_from_values(S, limit)
if name == "left_truncatable":
    S = set()
    for p in primes:
        s = str(p)
        if p <= 7:
            continue
        ok = True
        for i in range(len(s)):
            if int(s[i:]) not in pset:
                ok = False
                break
```

```
        if ok: S.add(p)
    return mask_from_values(S, limit)
if name == "right_truncatable":
    S = set()
    for p in primes:
        s = str(p)
        if p <= 7:
            continue
        ok = True
        for i in range(len(s)):
            if int(s[: len(s) - i]) not in pset:
                ok = False
                break
        if ok: S.add(p)
    return mask_from_values(S, limit)
if name in ("balanced", "strong", "weak"):
    S = set()
    for i in range(1, len(primes) - 1):
        a, b, c = primes[i - 1], primes[i], primes[i + 1]
        avg = (a + c) // 2
        if name == "balanced" and b * 2 == a + c: S.add(b)
        if name == "strong"   and b   > avg:      S.add(b)
        if name == "weak"     and b   < avg:      S.add(b)
    return mask_from_values(S, limit)
if name == "happy":
    return mask_from_values([p for p in primes if is_happy(p)], limit)
if name == "additive":
    sd_mask = prime_sieve(9 * int(math.log10(limit) + 1) + 10)
    sd_set  = set(np.flatnonzero(sd_mask))
    def sdigits(n):
        s = 0
        while n:
            n, d = divmod(n, 10)
            s += d
        return s
    return mask_from_values([p for p in primes if sdigits(p) in sd_set], limit)
if name == "factorial_pm1":
    S = set()
    f = 1
    n = 1
    while f - 1 <= limit or f + 1 <= limit:
        n += 1
        f *= n
        for cand in (f - 1, f + 1):
```

```
            if 2 <= cand <= limit and cand in pset:
                S.add(cand)
    return mask_from_values(S, limit)
if name == "primorial_pm1":
    S = set()
    f = 1
    for p in primes:
        f *= p
        for cand in (f - 1, f + 1):
            if 2 <= cand <= limit and cand in pset:
                S.add(cand)
        if f > limit * 3:
            break
    return mask_from_values(S, limit)
if name == "proth":
    S = set()
    for p in primes:
        m = p - 1
        n = 0
        while m % 2 == 0:
            m //= 2
            n += 1
        k = m
        if k % 2 == 1 and k < (1 << n):
            S.add(p)
    return mask_from_values(S, limit)
if name == "pierpont":
    S = set()
    for p in primes:
        r = p - 1
        while r % 2 == 0: r //= 2
        while r % 3 == 0: r //= 3
        if r == 1: S.add(p)
    return mask_from_values(S, limit)
if name == "wagstaff":
    S = set()
    small_primes = [q for q in primes if q < 64]
    for q in small_primes:
        if q % 2 == 0 or q < 3:
            continue
        num = (1 << q) + 1
        if num % 3 != 0:
            continue
        cand = num // 3
```

```
            if cand <= limit and cand in pset:
                S.add(cand)
        return mask_from_values(S, limit)
    if name == "superprime":
        idx_prime = set(np.flatnonzero(prime_sieve(len(primes) + 5)))
        S = set()
        for i, p in enumerate(primes, start=1):
            if i in idx_prime:
                S.add(p)
        return mask_from_values(S, limit)
    if name == "chen":
        S = set()
        upto = int(math.isqrt(limit + 2)) + 5
        small_mask = prime_sieve(upto)
        small_list = np.flatnonzero(small_mask)
        for p in primes:
            q = p + 2
            if q in pset or (q <= limit * 3 and is_semiprime(q, small_list)):
                S.add(p)
        return mask_from_values(S, limit)
    raise ValueError(f"Unknown prime filter: {name}")



# ------------------------------
# ULTRA-COMPLETE FUNCTION LIBRARY
# ------------------------------

# Precompute small factorials for speed
FACTORIAL_CACHE = np.array([special.factorial(i) for i in range(171)], dtype=np.float64)

@nb.njit
def fast_factorial(n):
    """Fast factorial with caching (up to 170!)"""
    if n < 0 or n > 170:
        return 0
    return FACTORIAL_CACHE[int(n)]

# Vectorized factorial
vfactorial = np.vectorize(fast_factorial)

# Prime checking (fast for small numbers)
@nb.njit
def is_prime(n):
```

```
    """Fast prime check"""
    if n < 2:
        return 0
    if n == 2:
        return 1
    if n % 2 == 0:
        return 0
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return 0
    return 1

vis_prime = np.vectorize(is_prime)

# Nth prime (cached up to n=10000)
PRIME_CACHE = np.array([i for i in range(2, 105000) if is_prime(i)])

def nth_prime(n):
    """Get nth prime (1-indexed)"""
    n = int(n)
    if n < 1 or n > len(PRIME_CACHE):
        return 0
    return PRIME_CACHE[n-1]

vnth_prime = np.vectorize(nth_prime)

# Fibonacci (fast)
@nb.njit
def fibonacci(n):
    """Fast fibonacci"""
    if n < 0:
        return 0
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(n-1):
        a, b = b, a + b
    return b

vfib = np.vectorize(fibonacci)

# Totient function
@nb.njit
def totient(n):
```

```
    """Euler's totient function φ(n)"""
    if n <= 0:
        return 0
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p
        p += 1
    if n > 1:
        result -= result // n
    return result

vtotient = np.vectorize(totient)

# Divisor count
@nb.njit
def divisor_count(n):
    """Count of divisors τ(n)"""
    if n <= 0:
        return 0
    count = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            count += 1 if i * i == n else 2
        i += 1
    return count

vdiv_count = np.vectorize(divisor_count)

# Divisor sum
@nb.njit
def divisor_sum(n):
    """Sum of divisors σ(n)"""
    if n <= 0:
        return 0
    total = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            total += i
```

```
            if i != n // i:
                total += n // i
        i += 1
    return total

vdiv_sum = np.vectorize(divisor_sum)


ALLOWED_FUNCS = {
    # ===== NUMPY CORE =====
    "np": np,

    # ===== BASIC ARITHMETIC =====
    "abs": np.abs,
    "round": np.round,
    "floor": np.floor,
    "ceil": np.ceil,
    "trunc": np.trunc,
    "sign": np.sign,

    # ===== ROOTS & POWERS =====
    "sqrt": np.sqrt,
    "cbrt": np.cbrt,
    "square": np.square,
    "power": np.power,
    "pow": np.power,

    # ===== EXPONENTIAL & LOG =====
    "exp": np.exp,
    "exp2": np.exp2,
    "expm1": np.expm1,
    "log": np.log,
    "log2": np.log2,
    "log10": np.log10,
    "log1p": np.log1p,

    # ===== TRIG =====
    "sin": np.sin,
    "cos": np.cos,
    "tan": np.tan,
    "arcsin": np.arcsin,
    "arccos": np.arccos,
    "arctan": np.arctan,
    "arctan2": np.arctan2,
```

```
"sinh": np.sinh,
"cosh": np.cosh,
"tanh": np.tanh,
"arcsinh": np.arcsinh,
"arccosh": np.arccosh,
"arctanh": np.arctanh,
"deg2rad": np.deg2rad,
"rad2deg": np.rad2deg,

# ===== NUMBER THEORY =====
"gcd": np.gcd,
"lcm": np.lcm,
"mod": np.mod,
"remainder": np.remainder,
"fmod": np.fmod,

# Number theory functions
"factorial": vfactorial,
"fac": vfactorial,
"fib": vfib,
"fibonacci": vfib,
"totient": vtotient,
"phi": vtotient,  # Euler's φ
"tau": vdiv_count,  # Divisor count
"sigma": vdiv_sum,  # Divisor sum
"is_prime": vis_prime,
"prime": vnth_prime,  # nth prime

# ===== BITWISE =====
"bitwise_and": np.bitwise_and,
"bitwise_or": np.bitwise_or,
"bitwise_xor": np.bitwise_xor,
"bitwise_not": np.bitwise_not,
"left_shift": np.left_shift,
"right_shift": np.right_shift,
"invert": np.invert,

# ===== CONDITIONALS =====
"where": np.where,
"minimum": np.minimum,
"maximum": np.maximum,
"clip": np.clip,
"select": np.select,
```

```
    # ===== SPECIAL FUNCTIONS =====
    "gamma": special.gamma,
    "loggamma": special.loggamma,
    "digamma": special.digamma,
    "erf": special.erf,
    "erfc": special.erfc,
    "bessel_j0": special.j0,
    "bessel_j1": special.j1,
    "bessel_y0": special.y0,
    "bessel_y1": special.y1,

    # ===== ARRAY OPERATIONS =====
    "sum": np.sum,
    "prod": np.prod,
    "mean": np.mean,
    "std": np.std,
    "var": np.var,
    "median": np.median,
    "cumsum": np.cumsum,
    "cumprod": np.cumprod,
    "diff": np.diff,
    "gradient": np.gradient,

    # ===== CONSTANTS =====
    "pi": np.pi,
    "PI": np.pi,
    "e": np.e,
    "E": np.e,
    "euler": np.euler_gamma,
    "golden": (1 + np.sqrt(5)) / 2,
    "phi_const": (1 + np.sqrt(5)) / 2,  # Golden ratio constant
    "sqrt2": np.sqrt(2),
    "sqrt3": np.sqrt(3),
    "sqrt5": np.sqrt(5),
}

# ===== SHORTHAND ALIASES =====
SHORTCUTS = {
    # Math shortcuts
    "S": np.sqrt,
    "SQ": np.square,
    "CB": np.cbrt,
    "LN": np.log,
    "EXP": np.exp,
```

```
    # Number theory shortcuts
    "G": np.gcd,
    "L": np.lcm,
    "M": np.mod,
    "F": vfactorial,
    "FIB": vfib,
    "TOT": vtotient,
    "DIV": vdiv_count,
    "SIG": vdiv_sum,
    "PR": vis_prime,
    "NP": vnth_prime,

    # Bitwise shortcuts
    "AND": np.bitwise_and,
    "OR": np.bitwise_or, # Non-exclusive OR
    "XOR": np.bitwise_xor,
    "NOT": np.bitwise_not,
    "LSH": np.left_shift,
    "RSH": np.right_shift,

    # Trig shortcuts
    "SIN": np.sin,
    "COS": np.cos,
    "TAN": np.tan,

    # Comparison shortcuts
    "MIN": np.minimum,
    "MAX": np.maximum,
}

# Merge shortcuts into main dict
ALLOWED_FUNCS.update(SHORTCUTS)


# ----------------------------
# SAFE EXPRESSION EVALUATOR
# ----------------------------
def evaluate_expression(N: int, expr: str, safe_mode: bool = True, clip_range: int = 10**15) ->
np.ndarray:
    """
    Evaluate ANY mathematical expression on NxN grid.

    Args:
```

```
    N: Grid size
    expr: Expression string (use x, y, any function from ALLOWED_FUNCS)
    safe_mode: Restrict to ALLOWED_FUNCS only
    clip_range: Maximum absolute value (prevents overflow)

Returns:
    Integer array (NxN)

Examples:
    >>> evaluate_expression(100, "gcd(x, y)")
    >>> evaluate_expression(100, "fib(x % 20) + fib(y % 20)")
    >>> evaluate_expression(100, "totient(x) ^ totient(y)")
    >>> evaluate_expression(100, "S(G(x, y)) * golden")
    >>> evaluate_expression(100, "OR(is_prime(x), is_prime(y))") # Example using OR
"""
# Create coordinate grids
x, y = np.indices((N, N), dtype=np.int64)

# Build environment
env = {"x": x, "y": y, **ALLOWED_FUNCS}

try:
    if safe_mode:
        val = eval(expr, {"__builtins__": {}}, env)
    else:
        val = eval(expr, {"__builtins__": {}, "np": np}, env)
except Exception as e:
    raise ValueError(f"Expression failed: {e}\n'{expr}'")

# Convert to array and handle special values
val = np.asarray(val, dtype=np.float64)
val = np.nan_to_num(val, nan=0.0, posinf=clip_range, neginf=-clip_range)

# Convert to integers
is_int = np.isfinite(val) & (np.abs(val - np.rint(val)) < 1e-9)
result = np.zeros_like(val, dtype=np.int64)
result[is_int] = np.rint(val[is_int]).astype(np.int64)

# Clip to prevent overflow
result = np.clip(result, -clip_range, clip_range)

return result
```

```
# ----------------------------
# EXPRESSION EXAMPLES
# ----------------------------
EXAMPLE_EXPRESSIONS = {
    "Basic GCD": "gcd(x, y)",
    "Nested operations": "sqrt(gcd(lcm(x**2, y), x*y))",
    "Fibonacci field": "fib(x % 20) + fib(y % 20)",
    "Totient geometry": "totient(x) ^ totient(y)",
    "Prime indicator": "is_prime(x) * is_prime(y)",
    "Divisor structure": "tau(x) + tau(y)",
    "Golden ratio grid": "floor(x * golden) ^ floor(y * golden)",
    "Factorial chaos": "mod(fac(x % 10), fac(y % 10) + 1)",
    "Bitwise primes": "XOR(prime(x % 100), prime(y % 100))",
    "Your monster": "S(G(L(x**2, y) + G(x, y**2), L(S(x*y), G(x**3, y))) ** L(G(x, S(y**2 + x)),
S(L(x**2, y**2))))",
}


# ----------------------------
# HELPER: List all available functions
# ----------------------------
def list_functions():
    """Print all available functions and shortcuts"""
    print("===== FULL FUNCTION LIST =====\n")

    categories = {
        "Basic Math": ["abs", "round", "floor", "ceil", "sqrt", "square", "power"],
        "Number Theory": ["gcd", "lcm", "mod", "factorial", "fibonacci", "totient", "tau", "sigma",
"is_prime", "prime"],
        "Trig": ["sin", "cos", "tan", "arcsin", "arccos", "arctan"],
        "Exponential": ["exp", "log", "log2", "log10"],
        "Bitwise": ["AND", "OR (non-exclusive)", "XOR", "NOT", "LSH", "RSH"],
        "Constants": ["pi", "e", "golden", "euler", "sqrt2", "sqrt3", "sqrt5"],
        "Shortcuts": ["S (sqrt)", "G (gcd)", "L (lcm)", "F (factorial)", "FIB", "TOT (totient)"],
    }

    for category, funcs in categories.items():
        print(f"\n{category}:")
        for func in funcs:
            print(f"  - {func}")

    print(f"\n\nTotal functions available: {len(ALLOWED_FUNCS)}")
```

```
if __name__ == "__main__":
    list_functions()

# ==================== ULTIMATE ANALYTICS ENGINE ====================

def analyze_fourier(field):
    """Enhanced Fourier analysis with directional decomposition."""
    f = field.astype(float)
    N = f.shape[0]

    # 2D FFT
    F = fft2(f)
    F_shifted = fftshift(F)

    mag = np.abs(F_shifted)
    phase = np.angle(F_shifted)
    power = mag ** 2
    log_mag = np.log1p(mag)

    # Dominant frequencies
    center = N // 2
    flat_idx = np.argsort(power.ravel())[::-1][:30]
    top_coords = np.unravel_index(flat_idx, power.shape)
    freq_x = top_coords[1] - center
    freq_y = top_coords[0] - center

    # Directional slope & angle
    mask = (freq_x != 0) | (freq_y != 0)
    fx, fy = freq_x[mask], freq_y[mask]
    if len(fx) > 1:
        slope, _ = np.polyfit(fx, fy, 1)
        angle_deg = np.degrees(np.arctan(slope))
    else:
        slope, angle_deg = 0, 0

    # Wavelengths
    dists = np.sqrt(freq_x**2 + freq_y**2)
    wavelengths = N / (dists + 1e-10)

    # Radial anisotropy
    Y, X = np.ogrid[:N, :N]
    r = np.sqrt((X - center)**2 + (Y - center)**2).astype(int)
    radial_mean = ndimage.mean(power, labels=r, index=np.arange(r.max() + 1))
    expected = radial_mean[r]
```

```
    anisotropy = np.std((power - expected) / (expected + 1e-10))

    # Angular power distribution
    theta = np.arctan2(Y - center, X - center)
    angular_bins = 16
    theta_binned = ((theta + np.pi) / (2 * np.pi) * angular_bins).astype(int) % angular_bins
    angular_power = ndimage.sum(power, labels=theta_binned, index=np.arange(angular_bins))
    angular_entropy = -np.sum((angular_power / angular_power.sum()) * np.log(angular_power /
angular_power.sum() + 1e-10))

    return {
        "fft_mag": log_mag,
        "fft_phase": phase,
        "fft_power": np.log1p(power),
        "top_freqs": (freq_x, freq_y),
        "top_powers": power[top_coords],
        "diagonal_slope": slope,
        "diagonal_angle_deg": angle_deg,
        "wavelengths": wavelengths[:8],
        "anisotropy": anisotropy,
        "angular_power": angular_power,
        "angular_entropy": angular_entropy,
    }


def analyze_topology(field):
    """Topological features: components, holes, boundaries."""
    # Connected components
    labeled, n_components = label(field)

    # Component sizes
    if n_components > 0:
        sizes = ndimage.sum(field, labeled, range(1, n_components + 1))
        sizes = np.array(sizes)
        avg_size = sizes.mean() if len(sizes) > 0 else 0
        max_size = sizes.max() if len(sizes) > 0 else 0
        size_std = sizes.std() if len(sizes) > 0 else 0
    else:
        avg_size = max_size = size_std = 0
        sizes = np.array([])

    # Euler characteristic approximation (V - E + F)
    # For 2D: χ = components - holes
    # Estimate holes by inverting and counting components
```

```
    inverted = 1 - field
    _, n_holes = label(inverted)
    euler_char = n_components - (n_holes - 1)  # -1 because background counts as one

    # Boundary length (perimeter)
    from scipy.ndimage import sobel
    sx = sobel(field.astype(float), axis=0)
    sy = sobel(field.astype(float), axis=1)
    boundary = np.hypot(sx, sy)
    boundary_length = np.sum(boundary > 0)

    # Distance transform for medial axis
    if np.any(field):
        dist = distance_transform_edt(field)
        skeleton_strength = dist.max()
    else:
        skeleton_strength = 0

    return {
        "n_components": n_components,
        "n_holes": max(0, n_holes - 1),
        "euler_characteristic": euler_char,
        "component_sizes": sizes,
        "avg_component_size": avg_size,
        "max_component_size": max_size,
        "size_std": size_std,
        "boundary_length": boundary_length,
        "skeleton_strength": skeleton_strength,
        "labeled_field": labeled,
    }


def analyze_graph(field, max_nodes=2000):
    """Graph analysis: treat primes as nodes, edges by proximity."""
    coords = np.argwhere(field > 0)
    n_nodes = len(coords)

    if n_nodes == 0:
        return {
            "n_nodes": 0, "n_edges": 0, "avg_degree": 0, "density": 0,
            "clustering": 0, "avg_path_length": 0, "diameter": 0,
            "degree_dist": [], "graph": None, "coords": coords,
        }
```

```
# Subsample if too many nodes
if n_nodes > max_nodes:
    idx = np.random.choice(n_nodes, max_nodes, replace=False)
    coords = coords[idx]
    n_nodes = max_nodes

# Build k-NN graph (k=8 for 2D grid connectivity)
k = min(8, n_nodes - 1)
if k < 1:
    return {
        "n_nodes": n_nodes, "n_edges": 0, "avg_degree": 0, "density": 0,
        "clustering": 0, "avg_path_length": 0, "diameter": 0,
        "degree_dist": [], "graph": None, "coords": coords,
    }

nbrs = NearestNeighbors(n_neighbors=k + 1, algorithm='kd_tree').fit(coords)
distances, indices = nbrs.kneighbors(coords)

# Build graph
G = nx.Graph()
G.add_nodes_from(range(n_nodes))

for i in range(n_nodes):
    for j, dist in zip(indices[i][1:], distances[i][1:]):  # skip self
        if dist < 5.0:  # proximity threshold
            G.add_edge(i, j, weight=dist)

n_edges = G.number_of_edges()

# Graph metrics
if n_nodes > 1:
    density = nx.density(G)
    degrees = [d for n, d in G.degree()]
    avg_degree = np.mean(degrees) if degrees else 0
    degree_dist = Counter(degrees)

    # Clustering coefficient
    try:
        clustering = nx.average_clustering(G)
    except:
        clustering = 0

    # Path metrics (expensive, use largest component)
    if nx.is_connected(G):
```

```
            largest_cc = G
        else:
            largest_cc = G.subgraph(max(nx.connected_components(G), key=len)).copy()

        if largest_cc.number_of_nodes() > 1:
            try:
                avg_path = nx.average_shortest_path_length(largest_cc)
                diameter = nx.diameter(largest_cc)
            except:
                avg_path = diameter = 0
        else:
            avg_path = diameter = 0
    else:
        density = avg_degree = clustering = avg_path = diameter = 0
        degree_dist = Counter()

    return {
        "n_nodes": n_nodes,
        "n_edges": n_edges,
        "avg_degree": avg_degree,
        "density": density,
        "clustering": clustering,
        "avg_path_length": avg_path,
        "diameter": diameter,
        "degree_dist": dict(degree_dist),
        "graph": G,
        "coords": coords,
    }


def analyze_autocorrelation(field):
    """2D autocorrelation & correlation length."""
    f = field.astype(float)
    N = f.shape[0]
    center = N // 2

    # Compute autocorrelation via FFT
    f_centered = f - f.mean()
    autocorr = np.fft.ifft2(np.abs(fft2(f_centered))**2).real
    autocorr = fftshift(autocorr)
    autocorr /= autocorr.max() if autocorr.max() > 0 else 1

    # Radial profile
    Y, X = np.ogrid[:N, :N]
```

```
    r = np.sqrt((X - center)**2 + (Y - center)**2).astype(int)
    radial_profile = ndimage.mean(autocorr, labels=r, index=np.arange(r.max() + 1))

    # Correlation length: where radial drops to 1/e
    threshold = np.exp(-1)
    corr_length_idx = np.where(radial_profile < threshold)[0]
    corr_length = corr_length_idx[0] if len(corr_length_idx) > 0 else N // 2

    return {
        "autocorr": autocorr,
        "radial_profile": radial_profile,
        "correlation_length": corr_length,
    }


def ultimate_analysis(field):
    """Combine all analytics."""
    fourier = analyze_fourier(field)
    topo = analyze_topology(field)
    graph = analyze_graph(field)
    autocorr = analyze_autocorrelation(field)

    return {
        "fourier": fourier,
        "topology": topo,
        "graph": graph,
        "autocorr": autocorr,
        "prime_density": float(field.mean()),
    }

# ==================== ULTIMATE UI (FIXED) ====================

def prime_field_lab():
    expr = widgets.Text(
        value="(x^2 + y^2)",
        description="Expression",
        layout=widgets.Layout(width="100%")
    )
    size = widgets.IntSlider(value=256, min=64, max=1024, step=64, description="Grid N")
    filters = widgets.SelectMultiple(
        options=ALL_FILTERS,
        value=("all",),
        description="Prime types",
        layout=widgets.Layout(height="180px", width="50%")
```

```
)

analysis_mode = widgets.Dropdown(
    options=["Full Analysis", "Fourier Only", "Graph Only", "Topology Only"],
    value="Full Analysis",
    description="Analysis"
)

out = widgets.Output()

def refresh(_=None):
    with out:
        clear_output(wait=True)
        plt.close('all')  # Close any existing figures

        N = size.value
        try:
            I = evaluate_expression(N, expr.value)
        except Exception as e:
            print(f"❌ Expression error: {e}")
            return

        valid = I >= 0
        if not np.any(valid):
            print("❌ No valid indices")
            return

        vmax = int(I[valid].max())
        if vmax < 2:
            vmax = 2

        base_prime_mask = prime_sieve(vmax)
        combined = np.zeros(vmax + 1, dtype=bool)
        for name in filters.value:
            combined |= get_filter_mask(name, base_prime_mask)

        field = np.zeros_like(I, dtype=np.uint8)
        ok = valid & (I <= vmax)
        field[ok] = combined[I[ok]].astype(np.uint8)

        # Run analysis
        mode = analysis_mode.value

        if mode == "Full Analysis":
```

```
stats = ultimate_analysis(field)

# Create mega-plot with proper GridSpec
fig = plt.figure(figsize=(24, 16))
gs = fig.add_gridspec(3, 4, hspace=0.3, wspace=0.3)

# Row 1: Field, FFT Mag, Power Spectrum, FFT Phase
ax1 = fig.add_subplot(gs[0, 0])
ax1.imshow(field, cmap="inferno", interpolation="nearest")
ax1.set_title(f"Prime Field\n{expr.value}", fontsize=10)
ax1.axis('off')

ax2 = fig.add_subplot(gs[0, 1])
ax2.imshow(stats["fourier"]["fft_mag"], cmap="viridis")
fx, fy = stats["fourier"]["top_freqs"]
ax2.scatter(fx + N//2, fy + N//2, c='red', s=30, alpha=0.7)
ax2.set_title("FFT Magnitude", fontsize=10)
ax2.axis('off')

ax3 = fig.add_subplot(gs[0, 2])
ax3.imshow(stats["fourier"]["fft_power"], cmap="hot")
ax3.set_title("Power Spectrum", fontsize=10)
ax3.axis('off')

ax3b = fig.add_subplot(gs[0, 3])
ax3b.imshow(stats["fourier"]["fft_phase"], cmap="twilight")
ax3b.set_title("FFT Phase", fontsize=10)
ax3b.axis('off')

# Row 2: Components, Autocorr, Angular Power, Graph
ax4 = fig.add_subplot(gs[1, 0])
labeled = stats["topology"]["labeled_field"]
ax4.imshow(labeled, cmap="inferno", interpolation="nearest")
ax4.set_title(f"Components: {stats['topology']['n_components']}", fontsize=10)
ax4.axis('off')

ax5 = fig.add_subplot(gs[1, 1])
ax5.imshow(stats["autocorr"]["autocorr"], cmap="Accent", vmin=-0.5, vmax=1)
ax5.set_title("Autocorrelation", fontsize=10)
ax5.axis('off')

ax6 = fig.add_subplot(gs[1, 2])
angular = stats["fourier"]["angular_power"]
ax6.bar(range(len(angular)), angular, color='cyan', edgecolor='black')
```

```
        ax6.set_title("Angular Power", fontsize=10)
        ax6.set_xlabel("Angle Bin", fontsize=8)

        ax7 = fig.add_subplot(gs[1, 3])
        G = stats["graph"]["graph"]
        coords = stats["graph"]["coords"]
        if G and len(coords) > 0:
            pos = {i: coords[i][::-1] for i in range(len(coords))}
            nx.draw(G, pos, node_size=1, node_color='Red', edge_color='blue',
                    width=0.3, alpha=0.6, ax=ax7)
            ax7.set_facecolor('black')
        ax7.set_title(f"Graph: {stats['graph']['n_nodes']} nodes", fontsize=10)
        ax7.axis('off')

        # Row 3: Degree Dist, Radial Profile, Freq Plot, Summary Boxes
        ax8 = fig.add_subplot(gs[2, 0])
        deg_dist = stats["graph"]["degree_dist"]
        if deg_dist:
            degrees = sorted(deg_dist.keys())
            counts = [deg_dist[d] for d in degrees]
            ax8.bar(degrees, counts, color='orange', edgecolor='black')
            ax8.set_yscale('log')
            ax8.set_title("Degree Distribution", fontsize=10)
            ax8.set_xlabel("Degree", fontsize=8)
            ax8.set_ylabel("Count", fontsize=8)

        ax9 = fig.add_subplot(gs[2, 1])
        radial = stats["autocorr"]["radial_profile"]
        ax9.plot(radial, color='purple', linewidth=2)
        ax9.axhline(np.exp(-1), color='red', linestyle='--', label='1/e')
        ax9.axvline(stats["autocorr"]["correlation_length"], color='green', linestyle='--',
label='Corr Length')
        ax9.set_title("Radial Autocorrelation", fontsize=10)
        ax9.legend(fontsize=7)
        ax9.grid(True, alpha=0.3)

        ax10 = fig.add_subplot(gs[2, 2])
        fx, fy = stats["fourier"]["top_freqs"]
        powers = np.log1p(stats["fourier"]["top_powers"])
        ax10.scatter(fx, fy, c=powers, cmap="plasma", s=100, edgecolor='white',
linewidth=0.5)
        ax10.axhline(0, color='white', linewidth=0.5, alpha=0.5)
        ax10.axvline(0, color='white', linewidth=0.5, alpha=0.5)
        slope = stats["fourier"]["diagonal_slope"]
```

```
            ax10.axline((0, 0), slope=slope, color='cyan', linestyle='--', linewidth=2)
            ax10.set_title(f"Freq Space (angle={stats['fourier']['diagonal_angle_deg']:.1f}°)",
fontsize=10)
            ax10.set_xlabel("freq_x", fontsize=8)
            ax10.set_ylabel("freq_y", fontsize=8)
            ax10.grid(True, alpha=0.3)

            # Summary box
            ax11 = fig.add_subplot(gs[2, 3])
            ax11.axis('off')
            combined_summary = f"""╔══ FOURIER ══╗
Density: {stats['prime_density']:.4f}
Angle: {stats['fourier']['diagonal_angle_deg']:.2f}°
Anisotropy: {stats['fourier']['anisotropy']:.4f}
Wavelengths:
{', '.join(f'{w:.1f}' for w in stats['fourier']['wavelengths'][:4])}

╔══ TOPOLOGY ══╗
Components: {stats['topology']['n_components']}
Holes: {stats['topology']['n_holes']}
Euler χ: {stats['topology']['euler_characteristic']}

╔══ GRAPH ══╗
Nodes: {stats['graph']['n_nodes']}
Clustering: {stats['graph']['clustering']:.4f}
Diameter: {stats['graph']['diameter']}
"""
            ax11.text(0.05, 0.95, combined_summary, transform=ax11.transAxes,
                    fontsize=8, verticalalignment='top', fontfamily='monospace',
                    bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.8))

            plt.show()

        elif mode == "Fourier Only":
            fourier = analyze_fourier(field)
            autocorr = analyze_autocorrelation(field)

            fig = plt.figure(figsize=(20, 10))
            gs = fig.add_gridspec(2, 4, hspace=0.3, wspace=0.3)

            ax1 = fig.add_subplot(gs[0, 0])
            ax1.imshow(field, cmap="inferno", interpolation="nearest")
            ax1.set_title(f"Prime Field", fontsize=12)
            ax1.axis('off')
```

```
ax2 = fig.add_subplot(gs[0, 1])
ax2.imshow(fourier["fft_mag"], cmap="viridis")
fx, fy = fourier["top_freqs"]
ax2.scatter(fx + N//2, fy + N//2, c='red', s=50, alpha=0.8)
ax2.set_title("FFT Magnitude (log)", fontsize=12)
ax2.axis('off')

ax3 = fig.add_subplot(gs[0, 2])
ax3.imshow(fourier["fft_phase"], cmap="twilight")
ax3.set_title("FFT Phase", fontsize=12)
ax3.axis('off')

ax4 = fig.add_subplot(gs[0, 3])
ax4.imshow(fourier["fft_power"], cmap="hot")
ax4.set_title("Power Spectrum (log)", fontsize=12)
ax4.axis('off')

ax5 = fig.add_subplot(gs[1, 0])
ax5.imshow(autocorr["autocorr"], cmap="Accent", vmin=-0.5, vmax=1)
ax5.set_title("2D Autocorrelation", fontsize=12)
ax5.axis('off')

ax6 = fig.add_subplot(gs[1, 1])
fx, fy = fourier["top_freqs"]
powers = np.log1p(fourier["top_powers"])
scatter = ax6.scatter(fx, fy, c=powers, cmap="plasma", s=150,
                      edgecolor='white', linewidth=1)
ax6.axhline(0, color='white', linewidth=1, alpha=0.5)
ax6.axvline(0, color='white', linewidth=1, alpha=0.5)
slope = fourier["diagonal_slope"]
ax6.axline((0, 0), slope=slope, color='cyan', linestyle='--', linewidth=3)
ax6.set_title(f"Frequency Space", fontsize=12)
ax6.set_xlabel("freq_x", fontsize=10)
ax6.set_ylabel("freq_y", fontsize=10)
ax6.grid(True, alpha=0.3)
plt.colorbar(scatter, ax=ax6, label="log(power)")

ax7 = fig.add_subplot(gs[1, 2])
angular = fourier["angular_power"]
bars = ax7.bar(range(len(angular)), angular, color='cyan',
               edgecolor='black', linewidth=1.5)
ax7.set_title("Angular Power Distribution", fontsize=12)
ax7.set_xlabel("Angle Bin (0=right, CCW)", fontsize=10)
```

```
            ax7.set_ylabel("Power", fontsize=10)
            ax7.grid(True, alpha=0.3, axis='y')

            ax8 = fig.add_subplot(gs[1, 3])
            radial = autocorr["radial_profile"]
            ax8.plot(radial, color='purple', linewidth=3, label='Radial Profile')
            ax8.axhline(np.exp(-1), color='red', linestyle='--', linewidth=2, label='1/e threshold')
            ax8.axvline(autocorr["correlation_length"], color='green',
                        linestyle='--', linewidth=2, label=f'ξ = {autocorr["correlation_length"]:.1f}')
            ax8.set_title("Radial Autocorrelation", fontsize=12)
            ax8.set_xlabel("Distance (pixels)", fontsize=10)
            ax8.set_ylabel("Correlation", fontsize=10)
            ax8.legend(fontsize=9)
            ax8.grid(True, alpha=0.3)
            ax8.set_ylim([-0.1, 1.1])

            plt.show()

            print(f"""
╔═══════════════════════════════════════╗
║     FOURIER ANALYSIS RESULTS          ║
╚═══════════════════════════════════════╝

📊 SPECTRAL METRICS:
  • Prime Density: {field.mean():.6f}
  • Dominant Angle: {fourier['diagonal_angle_deg']:.3f}°
  • Diagonal Slope: {fourier['diagonal_slope']:.6f}
  • Anisotropy Index: {fourier['anisotropy']:.6f}
  • Angular Entropy: {fourier['angular_entropy']:.6f}

🌊 WAVELENGTH SPECTRUM:
  Top 8: {', '.join(f'{w:.2f}' for w in fourier['wavelengths'][:8])}

🔗 CORRELATION:
  • Correlation Length ξ: {autocorr['correlation_length']:.2f} pixels
  • Decay Rate: {1/autocorr['correlation_length']:.4f} px⁻¹

🎯 INTERPRETATION:
  • Anisotropy > 0.5: Strong directional bias
  • Low Angular Entropy: Highly ordered structure
  • Large ξ: Long-range correlations present
""")

        elif mode == "Graph Only":
```

```
graph = analyze_graph(field, max_nodes=3000)

fig = plt.figure(figsize=(20, 12))
gs = fig.add_gridspec(2, 3, hspace=0.3, wspace=0.3)

ax1 = fig.add_subplot(gs[0, 0])
ax1.imshow(field, cmap="inferno", interpolation="nearest")
ax1.set_title(f"Prime Field", fontsize=12)
ax1.axis('off')

ax2 = fig.add_subplot(gs[0, 1])
G = graph["graph"]
coords = graph["coords"]
if G and len(coords) > 0:
    pos = {i: coords[i][::-1] for i in range(len(coords))}
    degrees = dict(G.degree())
    node_colors = [degrees[i] for i in G.nodes()]
    nx.draw(G, pos, node_size=5, node_color=node_colors,
            cmap='viridis', edge_color='yellow',
            width=0.5, alpha=0.7, ax=ax2, with_labels=False)
    ax2.set_facecolor('black')
ax2.set_title(f"Proximity Graph ({graph['n_nodes']} nodes)", fontsize=12)
ax2.axis('off')

ax3 = fig.add_subplot(gs[0, 2])
deg_dist = graph["degree_dist"]
if deg_dist:
    degrees = sorted(deg_dist.keys())
    counts = [deg_dist[d] for d in degrees]
    ax3.bar(degrees, counts, color='orange', edgecolor='black', linewidth=1.5)
    ax3.set_yscale('log')
    ax3.set_title("Degree Distribution", fontsize=12)
    ax3.set_xlabel("Degree k", fontsize=10)
    ax3.set_ylabel("Count (log scale)", fontsize=10)
    ax3.grid(True, alpha=0.3)

ax4 = fig.add_subplot(gs[1, 0])
if G and len(coords) > 0:
    degree_map = np.zeros_like(field, dtype=float)
    for i, coord in enumerate(coords):
        y, x = coord
        if i in degrees:
            degree_map[y, x] = degrees[i]
    im = ax4.imshow(degree_map, cmap="plasma", interpolation="nearest")
```

```
            ax4.set_title("Degree Centrality Map", fontsize=12)
            ax4.axis('off')
            plt.colorbar(im, ax=ax4, fraction=0.046)

        ax5 = fig.add_subplot(gs[1, 1])
        if G and G.number_of_nodes() > 0:
            clustering_coeffs = nx.clustering(G)
            if clustering_coeffs:
                coeffs = list(clustering_coeffs.values())
                ax5.hist(coeffs, bins=30, color='skyblue', edgecolor='black', alpha=0.7)
                ax5.axvline(np.mean(coeffs), color='red', linestyle='--',
                            linewidth=2, label=f'Mean: {np.mean(coeffs):.3f}')
                ax5.set_title("Clustering Coefficient Distribution", fontsize=12)
                ax5.set_xlabel("Clustering Coefficient", fontsize=10)
                ax5.set_ylabel("Count", fontsize=10)
                ax5.legend(fontsize=9)
                ax5.grid(True, alpha=0.3, axis='y')

        ax6 = fig.add_subplot(gs[1, 2])
        ax6.axis('off')
        summary = f"""╔══ GRAPH ANALYSIS ══╗

🔢 BASIC STATS:
  Nodes: {graph['n_nodes']}
  Edges: {graph['n_edges']}
  Density: {graph['density']:.20f}

📈 DEGREE METRICS:
  Avg Degree: {graph['avg_degree']:.3f}
  Max Degree: {max(graph['degree_dist'].keys()) if graph['degree_dist'] else 0}

🕸️ STRUCTURE:
  Clustering: {graph['clustering']:.6f}
  Avg Path: {graph['avg_path_length']:.3f}
  Diameter: {graph['diameter']}

🧠 INTERPRETATION:
  High clustering + low diam
  → Small-world network

  Power-law degree dist
  → Scale-free network
"""
            ax6.text(0.05, 0.95, summary, transform=ax6.transAxes,
```

```
            fontsize=10, verticalalignment='top', fontfamily='monospace',
            bbox=dict(boxstyle='round', facecolor='lightyellow', alpha=0.9))

    plt.show()

elif mode == "Topology Only":
    topo = analyze_topology(field)

    fig = plt.figure(figsize=(20, 10))
    gs = fig.add_gridspec(2, 4, hspace=0.3, wspace=0.3)

    ax1 = fig.add_subplot(gs[0, 0])
    ax1.imshow(field, cmap="inferno", interpolation="nearest")
    ax1.set_title(f"Prime Field", fontsize=12)
    ax1.axis('off')

    ax2 = fig.add_subplot(gs[0, 1])
    labeled = topo["labeled_field"]
    ax2.imshow(labeled, cmap="tab20", interpolation="nearest")
    ax2.set_title(f"Connected Components ({topo['n_components']})", fontsize=12)
    ax2.axis('off')

    ax3 = fig.add_subplot(gs[0, 2])
    inverted = 1 - field
    ax3.imshow(inverted, cmap="bone", interpolation="nearest")
    ax3.set_title(f"Holes/Voids ({topo['n_holes']})", fontsize=12)
    ax3.axis('off')

    ax4 = fig.add_subplot(gs[0, 3])
    from scipy.ndimage import sobel
    sx = sobel(field.astype(float), axis=0)
    sy = sobel(field.astype(float), axis=1)
    boundary = np.hypot(sx, sy)
    ax4.imshow(boundary, cmap="hot", interpolation="nearest")
    ax4.set_title(f"Boundary (len={topo['boundary_length']:.0f})", fontsize=12)
    ax4.axis('off')

    ax5 = fig.add_subplot(gs[1, 0])
    if np.any(field):
        dist = distance_transform_edt(field)
        im = ax5.imshow(dist, cmap="coolwarm", interpolation="nearest")
        ax5.set_title("Distance Transform", fontsize=12)
        ax5.axis('off')
        plt.colorbar(im, ax=ax5, fraction=0.046)
```

```
            ax6 = fig.add_subplot(gs[1, 1])
            sizes = topo["component_sizes"]
            if len(sizes) > 0:
                ax6.hist(sizes, bins=min(50, len(sizes)), color='green',
                         edgecolor='black', alpha=0.7)
                ax6.set_yscale('log')
                ax6.set_title("Component Size Distribution", fontsize=12)
                ax6.set_xlabel("Size (pixels)", fontsize=10)
                ax6.set_ylabel("Count (log)", fontsize=10)
                ax6.grid(True, alpha=0.3, axis='y')

            ax7 = fig.add_subplot(gs[1, 2:])
            ax7.axis('off')
            summary = f"""╔═══════ TOPOLOGICAL ANALYSIS ═══════╗

🔗 CONNECTIVITY:
  Components: {topo['n_components']}
  Holes: {topo['n_holes']}
  Euler Characteristic χ: {topo['euler_characteristic']}

📏 SIZE STATISTICS:
  Average Component: {topo['avg_component_size']:.2f} px
  Largest Component: {topo['max_component_size']:.0f} px
  Size Std Dev: {topo['size_std']:.2f} px

🎯 MORPHOLOGY:
  Boundary Length: {topo['boundary_length']:.0f} px
  Skeleton Strength: {topo['skeleton_strength']:.2f}
  Perimeter/Area: {topo['boundary_length'] / (field.sum() + 1e-10):.4f}

🧱 TOPOLOGICAL INVARIANTS:
  • χ = V - E + F (Euler characteristic)
  • χ > 0: More components than holes
  • χ < 0: More holes than components
  • χ = 0: Balanced structure

📊 FRACTAL METRICS:
  • High boundary/area: Complex, fractal-like
  • Low boundary/area: Smooth, blob-like

Current: {topo['boundary_length'] / (field.sum() + 1e-10):.4f}
"""
            ax7.text(0.05, 0.95, summary, transform=ax7.transAxes,
```

```
                fontsize=10, verticalalignment='top', fontfamily='monospace',
                bbox=dict(boxstyle='round', facecolor='lightcyan', alpha=0.9))

        plt.show()

    expr.observe(refresh, "value")
    size.observe(refresh, "value")
    filters.observe(refresh, "value")
    analysis_mode.observe(refresh, "value")

    display(widgets.VBox([expr, size, filters, analysis_mode, out]))
    refresh()

# LAUNCH THE BEAST
prime_field_lab()



import React, { useState, useEffect, useRef } from 'react';

const XORPrimeFFT = () => {
  const [gridSize, setGridSize] = useState(400);

  const xorCanvasRef = useRef(null);
  const primeCanvasRef = useRef(null);
  const fftCanvasRef = useRef(null);
  const ifft1CanvasRef = useRef(null);
  const ifft2CanvasRef = useRef(null);

  const [stats, setStats] = useState('');

  // Fast prime check with cache
  const primeCache = useRef(new Map());
  const isPrime = (n) => {
    if (n < 2) return false;
    if (primeCache.current.has(n)) return primeCache.current.get(n);
    if (n === 2) { primeCache.current.set(n, true); return true; }
    if (n % 2 === 0) { primeCache.current.set(n, false); return false; }
    const sqrt = Math.sqrt(n);
    for (let i = 3; i <= sqrt; i += 2) {
      if (n % i === 0) { primeCache.current.set(n, false); return false; }
    }
    primeCache.current.set(n, true);
    return true;
```

```
};

// Fast 1D FFT
const fft = (re, im, inv = false) => {
  const n = re.length;
  if (n === 1) return;

  const halfN = n / 2;
  const reEven = new Float64Array(halfN);
  const imEven = new Float64Array(halfN);
  const reOdd = new Float64Array(halfN);
  const imOdd = new Float64Array(halfN);

  for (let i = 0; i < halfN; i++) {
    reEven[i] = re[2 * i];
    imEven[i] = im[2 * i];
    reOdd[i] = re[2 * i + 1];
    imOdd[i] = im[2 * i + 1];
  }

  fft(reEven, imEven, inv);
  fft(reOdd, imOdd, inv);

  const sign = inv ? 1 : -1;
  for (let k = 0; k < halfN; k++) {
    const angle = sign * 2 * Math.PI * k / n;
    const cos = Math.cos(angle);
    const sin = Math.sin(angle);

    const tRe = reOdd[k] * cos - imOdd[k] * sin;
    const tIm = reOdd[k] * sin + imOdd[k] * cos;

    re[k] = reEven[k] + tRe;
    im[k] = imEven[k] + tIm;
    re[k + halfN] = reEven[k] - tRe;
    im[k + halfN] = imEven[k] - tIm;

    if (inv) {
      re[k] /= 2;
      im[k] /= 2;
      re[k + halfN] /= 2;
      im[k + halfN] /= 2;
    }
  }
```

```
};

const fft2D = (data, N, inverse = false) => {
  const re = new Array(N);
  const im = new Array(N);

  // Initialize from magnitude data
  for (let i = 0; i < N; i++) {
    re[i] = new Float64Array(N);
    im[i] = new Float64Array(N);
    for (let j = 0; j < N; j++) {
      re[i][j] = data[i * N + j];
      im[i][j] = 0;
    }
  }

  // Transform rows
  for (let i = 0; i < N; i++) {
    fft(re[i], im[i], inverse);
  }

  // Transform columns
  for (let j = 0; j < N; j++) {
    const colRe = new Float64Array(N);
    const colIm = new Float64Array(N);
    for (let i = 0; i < N; i++) {
      colRe[i] = re[i][j];
      colIm[i] = im[i][j];
    }
    fft(colRe, colIm, inverse);
    for (let i = 0; i < N; i++) {
      re[i][j] = colRe[i];
      im[i][j] = colIm[i];
    }
  }

  // Compute magnitude
  const mag = new Float64Array(N * N);
  for (let i = 0; i < N; i++) {
    for (let j = 0; j < N; j++) {
      mag[i * N + j] = Math.sqrt(re[i][j] ** 2 + im[i][j] ** 2);
    }
  }
```

```
  return mag;
};

const drawCanvas = (canvas, data, N, isPrimeMask = false) => {
  if (!canvas) return;

  const ctx = canvas.getContext('2d', { alpha: false });
  canvas.width = N;
  canvas.height = N;

  const imgData = ctx.createImageData(N, N);
  const pixels = imgData.data;

  let min = Infinity, max = -Infinity;
  for (let i = 0; i < N * N; i++) {
    if (data[i] < min) min = data[i];
    if (data[i] > max) max = data[i];
  }

  const range = max - min || 1;

  for (let i = 0; i < N * N; i++) {
    const val = (data[i] - min) / range;
    const idx = i * 4;

    if (isPrimeMask) {
      if (data[i] > 0) {
        pixels[idx] = 255;
        pixels[idx + 1] = 255;
        pixels[idx + 2] = 0;
      } else {
        pixels[idx] = 0;
        pixels[idx + 1] = 0;
        pixels[idx + 2] = 0;
      }
    } else {
      const r = Math.round(255 * Math.min(1, val * 1.5));
      const g = Math.round(255 * Math.min(1, val * 1.2));
      const b = Math.round(255 * Math.min(1, val * 0.8));
      pixels[idx] = r;
      pixels[idx + 1] = g;
      pixels[idx + 2] = b;
    }
    pixels[idx + 3] = 255;
```

```
  }

  ctx.putImageData(imgData, 0, 0);
};

const compute = () => {
  const N = gridSize;
  const startTime = performance.now();

  // 1. XOR field
  const xorField = new Float64Array(N * N);
  for (let y = 0; y < N; y++) {
    for (let x = 0; x < N; x++) {
      xorField[y * N + x] = x ^ y;
    }
  }
  drawCanvas(xorCanvasRef.current, xorField, N);

  // 2. Prime mask
  const primeField = new Float64Array(N * N);
  let primeCount = 0;
  for (let i = 0; i < N * N; i++) {
    if (isPrime(xorField[i])) {
      primeField[i] = 1;
      primeCount++;
    }
  }
  drawCanvas(primeCanvasRef.current, primeField, N, true);

  // 3. FFT on primes
  const fftMag = fft2D(primeField, N, false);
  const fftLog = new Float64Array(N * N);
  for (let i = 0; i < N * N; i++) {
    fftLog[i] = Math.log(1 + fftMag[i]);
  }
  drawCanvas(fftCanvasRef.current, fftLog, N);

  // 4. IFFT on FFT result (inverting frequency space itself)
  const ifft1Mag = fft2D(fftLog, N, true);
  drawCanvas(ifft1CanvasRef.current, ifft1Mag, N);

  // 5. IFFT² - inverse of the inverse
  const ifft2Mag = fft2D(ifft1Mag, N, true);
  drawCanvas(ifft2CanvasRef.current, ifft2Mag, N);
```

```
    const elapsed = (performance.now() - startTime).toFixed(0);
    setStats(`${N}×${N} | ${primeCount} primes (${(primeCount/(N*N)*100).toFixed(1)}%) |
${elapsed}ms`);
  };

  useEffect(() => {
    compute();
  }, [gridSize]);

  return (
    <div className="w-full min-h-screen bg-black p-2 text-white">
      <div className="max-w-7xl mx-auto">
        <div className="flex justify-between items-center mb-2">
          <h1 className="text-2xl font-bold">XOR → PRIME → FFT → IFFT → IFFT²</h1>
          <div className="text-green-400 font-mono text-sm">{stats}</div>
        </div>

        <div className="flex gap-2 mb-2">
          <select
            value={gridSize}
            onChange={(e) => setGridSize(parseInt(e.target.value))}
            className="bg-gray-800 p-2 rounded text-sm"
          >
            <option value="128">128×128</option>
            <option value="256">256×256</option>
            <option value="400">400×400</option>
            <option value="512">512×512</option>
          </select>

          <button
            onClick={compute}
            className="bg-purple-600 hover:bg-purple-700 px-4 py-2 rounded font-bold text-sm"
          >
            🔥 COMPUTE NOW
          </button>
        </div>

        <div className="grid grid-cols-5 gap-1">
          <div>
            <div className="text-xs text-center mb-1">XOR(x,y)</div>
            <canvas ref={xorCanvasRef} className="w-full border border-gray-700"
style={{imageRendering: 'pixelated'}} />
          </div>
```

```
        <div>
          <div className="text-xs text-center mb-1 text-yellow-400">PRIMES ONLY</div>
          <canvas ref={primeCanvasRef} className="w-full border border-yellow-600"
style={{imageRendering: 'pixelated'}} />
        </div>

        <div>
          <div className="text-xs text-center mb-1">FFT (freq)</div>
          <canvas ref={fftCanvasRef} className="w-full border border-gray-700"
style={{imageRendering: 'pixelated'}} />
        </div>

        <div>
          <div className="text-xs text-center mb-1 text-cyan-400">IFFT¹ (inv freq)</div>
          <canvas ref={ifft1CanvasRef} className="w-full border border-cyan-600"
style={{imageRendering: 'pixelated'}} />
        </div>

        <div>
          <div className="text-xs text-center mb-1 text-purple-400">IFFT² (inv inv)</div>
          <canvas ref={ifft2CanvasRef} className="w-full border border-purple-600"
style={{imageRendering: 'pixelated'}} />
        </div>
       </div>

       <div className="mt-4 text-sm text-gray-400">
         <p><strong className="text-white">What we're doing:</strong></p>
         <p>1. XOR field → 2. Extract PRIMES → 3. FFT (frequency domain) → 4. IFFT on
frequency data → 5. IFFT on inverted frequency data</p>
         <p className="mt-2 text-cyan-400">We're inverting the spectral space recursively to see
the underlying structure</p>
       </div>
     </div>
   </div>
  );
};

export default XORPrimeFFT;
```